How to ConQueR Why-Not Questions

Quoc Trung Tran and Chee-Yong Chan
Department of Computer Science, School of Computing
National University of Singapore
{tqtrung, chancy}@comp.nus.edu.sg

ABSTRACT

One useful feature that is missing from today's database systems is an explain capability that enables users to seek clarifications on unexpected query results. There are two types of unexpected query results that are of interest: the presence of unexpected tuples, and the absence of expected tuples (i.e., missing tuples). Clearly, it would be very helpful to users if they could pose follow-up *why* and *why-not* questions to seek clarifications on, respectively, unexpected and expected (but missing) tuples in query results. While the why questions can be addressed by applying established data provenance techniques, the problem of explaining the why-not questions has received very little attention. There are currently two explanation models proposed for why-not questions. The first model explains a missing tuple t in terms of modifications to the database such that t appears in the query result wrt the modified database. The second model explains by identifying the data manipulation operator in the query evaluation plan that is responsible for excluding t from the result. In this paper, we propose a new paradigm for explaining a why-not question that is based on automatically generating a refined query whose result includes both the original query's result as well as the user-specified missing tuple(s). In contrast to the existing explanation models, our approach goes beyond merely identifying the "culprit" query operator responsible for the missing tuple(s) and is useful for applications where it is not appropriate to modify the database to obtain missing tuples.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Query processing; H.2.8 [**Database Management**]: Database Applications

General Terms

Algorithms, Design

SIGMOD'10, June 6–11, 2010, Indianapolis, Indiana, USA.

Keywords

query explanation, query refinement, why-not questions

1. INTRODUCTION

While database system research has made tremendous advances on functionality and performance related issues over the years, research on improving database usability has not attracted as much attention as it deserves [8]. One useful feature that is missing from today's database systems is an explain capability for users to seek clarifications on query results. Although most database systems today provide an explain functionality to help database administrators understand and tune the performance of unexpected slow-running queries, there is no similar higher-level explain feature available to help end users understand the unexpected results in their query outputs. There are two types of unexpected query results that are of interest: (1) the presence of unexpected tuples, and (2) the absence of expected tuples (i.e., missing tuples). Clearly, it would be very helpful if users could pose follow-up *why* questions (i.e., why is a certain tuple in the result) or *why-not* questions (i.e., why is a certain tuple missing from the result) to seek clarifications on unexpected query results. While the why questions can be addressed by applying established data provenance techniques [13], the problem of explaining the why-not questions has received very little attention [2].

Consider the following SQL query to find the recent high-scoring NBA players from the NBA statistics[1]: SELECT P.name FROM Player P, Regular R WHERE P.pID = R.pID AND R.year > 2000 AND R.pts > 2400. Among the players returned by the query are many expected well-known NBA superstars such as "LeBron James" and "Kobe Bryant". However, the user is surprised to find that the superstar "Rick Barry" is absent from the query result. At this point, the user could try to figure out for himself an explanation for the missing tuple by relaxing at least one selection predicate (e.g., adjusting the year to 1990 or lowering the points to 2000) to see if Barry satisfies the revised query. Clearly, such a manual trial-and-error approach of seeking explanation is rather tedious involving possibly many rounds of query refinement. Moreover, the user could end up over-relaxing his refined query and obtaining many more additional result tuples than just the tuple for Barry.

Thus, it would be very helpful to the user if she could simply pose a single why-not question to the database system to seek an explanation for why "Rick Barry" is not in the

[1] http://www.basketballreference.com/

result. There are two main models for explaining why-not questions. One natural explanation model for missing tuple(s) is to identify the query operator(s) that is responsible for eliminating the missing tuple(s) from the result [2]. Thus, for the above example, a possible explanation is to identify the selection operator on the year attribute as the "culprit" operator. For applications where a query result is computed by a workflow of black-box processing steps, the ability to pinpoint the step that is responsible for the missing tuple(s) could be the most informative available explanation.

However, in general, an even more helpful explanation can go beyond merely identifying the culprit step/operator and actually suggests one or more ways to "fix" the original query such that the missing tuple(s) become present in the result. Continuing with the example, a more informative explanation would be a refined query that changes the selection predicate on year to "R.year $>$ 1970". In this way, not only does the system reveal the culprit operator to the user, it also explicitly shows the user how to revise the original query to obtain the expected tuple(s). The automatic generation of refined queries to explain unexpected query results can be useful even for applications that interact with users via a form-based web-interface, where the SQL queries being issued to the database are generated by a middleware component based on the completed forms. Basically, what is needed is a component to map a refined query back to an interface-based explanation. For example, an interface-based explanation could inform the user that had she clicked on button X on the form and selected item Y from the pulled-down menu, the expected missing tuple would have been included in the result.

A second model that has been proposed explains a missing tuple t in terms of modifications to the database such that t appears in the query result wrt the modified database [7]. This model was proposed in the context where some of the data in the database are extracted from untrusted information sources which may not be accurate. Thus, the intuition of this model is to explain in terms of how to modify some of the untrusted data in order to produce the missing tuple. Clearly, this explanation model is very flexible if arbitrary modifications to the database are allowed to derive the missing tuple(s). However, this model may not be applicable in applications where all the data stored are trusted (e.g., enterprise databases) and where it may not be meaningful to make arbitrary changes to the stored data.

In this paper, we propose a new explanation model that is based on automatically generating one or more *refined query* whose result includes both the original query's result as well as the missing tuple(s). Our proposed model goes beyond identifying culprit query operator(s) (in contrast to the first explanation model), and actually recommends refined queries, instead of data changes (unlike the second explanation model), to "fix" missing tuple(s). While it is desirable for a refined query to be as similar as possible to the original input query by making only minimal relaxations to appropriate selection predicates in the query, doing so might not always generate the desired missing tuple(s) as the user's original query might actually be focused on the "wrong" part of the database schema and needs to be reformulated. Thus, our proposed explanation strategy will try to generate minimally modified refined queries to account for the missing tuple(s) whenever possible and resort to more drastic query reformulation if minimally refined queries do not exist. Besides handling why-not questions for select-project-join (SPJ) queries, our approach can also explain why-not questions for SPJ queries with aggregation (SPJA queries) which are not addressed by any of the existing explanation models.

The following three examples illustrate the capabilities of our proposed approach. The first example illustrates the need to sometimes refine query beyond simply relaxing selection predicates by reformulating the query to retrieve from different relations in the database schema.

EXAMPLE 1. *Consider a flight database which includes two relations budget_airline and business_airline that describe airfare information for budget and business airlines, respectively. Suppose a user wants to buy a cheap airticket for vacation travel in July with the criteria that the departure city is in New York, the maximum number of hops is at most* 3*, and the ticket price is at most* 1000*. The user issues a query on the budget_airline relation to find all the destination cities that meet his requirement and is surprised to learn that Shanghai is not listed in the result even though one of his colleagues has recently booked a cheap airticket to Shanghai. It could well be that there are no available air tickets from New York to Shanghai with the budget airlines but there are promotion cheap airtickets available with the business airlines that meet his requirement. Thus, in this case, simply relaxing the predicates in the original input query would not help to generate any explanation; instead, the refined query needs to be reformulated on both the budget_airline and business_airline relations.* □

The next example illustrates why-not queries with constraints on aggregated values.

EXAMPLE 2. *Consider the following query to find the average points scored by high-scoring recent basketball players: SELECT P.name, AVG(R.pts) FROM Player P, Regular R WHERE P.pID = R.pID AND R.pts* $>$ 2000 *AND R.year* $\geq$ 1994 *AND R.year* $\leq$ 2000 *GROUP BY P.name. The result contains two tuples (Michael Jordan,* 2200*) and (Gary Payton,* 2800*). The user might be expecting Jordan's average score to be higher and would like to seek an explanation for why Jordan's average score is not higher than* 3000*. Our approach can process such why-not questions involving selection constraints on an aggregated value. It turns out that Jordan actually did not perform so well during 1994, and a refined query that replaces "R.year* $\geq$ 1994*" with "R.year* $\geq$ 1995*" would explain the user's why-not question. Our approach can also support this kind of why-not questions for missing tuples. For example, the user could ask why is "Wilt Chamberlain" not in the result? Or even more specifically, ask why is "Wilt Chamberlain" not in the result with an average score of at least* 3000*?* □

The final example illustrates more complex why-not queries involving relative comparisons of aggregated values.

EXAMPLE 3. *Suppose Professor P issues the following query to check the academic performance (in terms of average scores) of his students: SELECT G.name, AVG (G.score) FROM Grade G GROUP BY G.name. P is surprised to find that the tuples (Alice,*70*) and (Bob,* 90*) in the result as he has expected Alice to perform better than Bob. Thus, P would like to ask why Alice's average score is not higher than Bob's.*

Our approach can handle such sophisticated why-not questions that involve comparisons among multiple aggregated values in the results. A possible explanation for this why-not question is the following refined query: SELECT G.name, AVG (G.score) FROM Grade G WHERE G.dept = "CS" GROUP BY G.name, which explains that Alice indeed performs better than Bob if the average scores were computed for courses offered by the CS department. □

Contributions. In this paper, we introduce a new framework, named `ConQueR`, to explain why-not questions based on automatically generating refined queries. We propose novel algorithms to not only handle basic SPJ queries but also more sophisticated SPJA queries that involve constraints or comparisons among aggregated values. We demonstrate the usefulness of our paradigm by comparing against the two existing explanation models on both synthetic and real data sets, and show the efficiency of our proposed algorithms by a performance comparison against an existing classification-based approach for generating instance-equivalent queries.

2. RELATED WORK

There are two existing models to explain why-not questions on query results. The first approach explains by modifying some tuples in the database so that the result of the query on the modified database will include both the original result and the specified missing tuples [7]. This work is orthogonal to our approach, which is based on modifying the input query. Unlike our work, [7] focuses only on SPJ queries and does not address why-not questions on SPJ queries with aggregation (SPJA queries).

The recent work, called Artemis [6], extends [7] by supporting a set of why-not tuples over a set of SPJU (SPJ with union) queries with constraints on minimizing the number of inserted tuples and constraints among why-not tuples using variables. Our work can also handle SPJU queries with constraints on why-not tuples. Similar to [7], Artemis does not handle SPJA queries.

The second approach, introduced in [2], explains missing tuples by identifying the manipulation operation(s) in the query plan that are responsible for excluding the missing tuples. The work here also focuses only on SPJ queries and does not consider SPJA queries. The idea of identifying the "culprit" operator to explain unexpected query results also appears in [5] in the context of explaining mismatches in schema matching.

The recent work called TALOS [15] for generating instance-equivalent queries for an input query is similar in spirit to our query refinement approach, and can be applied to our problem by treating the original query result and the missing tuples collectively as a query's output. Indeed, we have extended TALOS as an alternative solution for this work. Our experimental results reveal that TALOS is a more precision-oriented approach as the queries generated can be rather different from the input query. In some applications, it may not be too meaningful to explain missing tuples using refined queries that are very different from the input query. Moreover, the performance of TALOS is also slower than our approach by up to factor of 6 times due to its costly data classification step.

There is also some related work on query refinement to modify an input query so that its query result can satisfy some cardinality constraints. The work in [9, 11] relaxes the *failed* queries which return empty result so that the modified queries will yield some answers. As the goal there is to refine the query to return any non-empty result, the techniques there cannot be applied to our problem which has stronger constraints to satisfy. Another related direction in [10, 3] deals with the problem when a query returns too many/few answers by refining the query to satisfy some constraint on the query result size. Similar to the work in [9], the focus there is on the size of the output but not on the content of the output.

Another related direction is the work on provenance [4] and OLAP [12]. The work in [4] can trace the provenance of an aggregated value by finding the data that derived a given aggregated value; however, the techniques cannot be extended to handle the why-not questions that we address for SPJA queries. The work in [12] addresses explanations for OLAP applications to explain why an aggregated value in a data cube cell is lower/higher than the value in another cell. The main focus there is to compute a compact summarization of the data tuples at the detailed lower levels to account for the phenomena. Again, the work there cannot be generalized to solve our problem related to complex why-not questions on SPJA queries.

3. OVERVIEW OF OUR APPROACH

In this paper, we consider two fragments of SQL queries where each projected attribute is either a relation's attribute or a value computed by an aggregation operator (COUNT, SUM, or AVG) that does not involve any arithmetic expression in the operator's argument. The first fragment is the basic select-project-join (SPJ) queries where the selection condition is a conjunction of predicates $C_1 \wedge \cdots \wedge C_\ell$. Each C_i is either a selection predicate "A_j *op* c" or a join predicate "A_j *op* A_k", where A_i is an attribute, c is a constant, and *op* is a comparison operator. The second fragment is SPJ queries with aggregation (SPJA queries) which are SPJ SQL queries with aggregation operators in the select-clause and an optional group-by clause.

For simplicity and without loss of generality, we assume all attributes are numerical and consider only the "$\leq$" comparison operator for selection predicates. Our approach can be easily extended to categorical attributes and other comparison operators.

Running example. In this paper, we use the NBA statistics database on basketball players as our running example which consists of three main tables. The *Player* relation contains the identifier (*pID*) and name (*name*) of each player. The *Regular* (resp. *Playoff*) relation provides the number of points (*pts*), steal (*stl*), block (*blk*) and rebound (*reb*) statistics of a player when he was playing for a team (*team*) in a specific year (*year*) in regular season (resp. playoff) games. Figure 1 shows our running example data.

3.1 Why-not Questions & Refined Queries

Given an input query Q on a database D, let $Q(D)$ denote the output of Q on D. In the most basic form, a *why-not question* on $Q(D)$ is represented by a non-empty set of *why-not tuples* $S = \{t_1, \cdots, t_n\}$, $n \geq 1$, where each why-not tuple t_i has the same schema as Q and $t_i \notin Q(D)$. Essentially, the why-not question is asking why S is not a subset of $Q(D)$; i.e., why each $t_i \in S$ is not in $Q(D)$. Each component value in a why-not tuple can be in one of three

pID	*name*
P1	"A"
P2	"B"
P3	"C"
P4	"D"
P5	"E"

(a) *Player*

pID	team	year	*pts*	*blk*	*stl*	*reb*
P1	GSW	1973	2009	30	150	40
P2	SEA	1994	1689	35	200	281
P2	SEA	1995	1563	50	240	339
P3	CHI	1992	2541	45	220	361
P4	LAL	1995	1567	30	162	359

(b) *Regular*

pID	team	year	*pts*	*blk*	*stl*	*reb*
P1	GSW	1973	2029	40	100	30
P2	SEA	1994	3000	65	150	181
P2	CHI	1995	2200	50	120	161
P2	LAL	1996	2500	70	110	200
P4	LAL	1995	2300	70	150	150
P5	DEN	2000	1689	35	200	381

(c) *Playoff*

Figure 1: Running Example: Basketball Data Set D

forms: (1) a constant value compatible with the corresponding attribute's domain; (2) a don't-care value (denoted by _); or (3) a variable (denoted by a $ symbol followed by a sequence of letters; e.g., $\$x$). A don't-care value is used for an attribute A_i when the user is not interested in the specific value of attribute A_i in the why-not tuple. A variable is used for an attribute A_i when the user wishes to impose a selection condition on that attribute in the why-not tuple with respect to some constant value or another attribute appearing in the same or another why-not tuple as illustrated by the SPJA queries in Examples 2 and 3. Thus, in the most general form, a why-not question on $Q(D)$ is represented by (S, C) where S is a non-empty set of why-not tuples and C is a (possibly empty) set of selection conditions defined on the variables appearing in S. In Example 3, the why-not question is represented by $S = \{(\text{Alice},\$x), (\text{Bob},\$y)\}$ and $C = \{\$x > \$y\}$.

Given a why-not question (S, C) on $Q(D)$, we say that Q' is a *refined query* of Q that explains the why-not question (S, C) if (1) $Q'(D)$ contains $Q(D)$, and (2) for each why-not tuple $t_w \in S$, there exists a *matching tuple* $t \in Q'(D)$ such that all the constraints in C are satisfied by the matching tuples. A tuple $t \in Q'(D)$ is a *matching tuple* for a why-not $t_w \in S$ if for every component of t_w that is a constant value, the corresponding component in t has the same constant value. Thus, if t is a matching tuple for t_w, then every component of t_w that is a variable becomes instantiated with the corresponding attribute value in t, and the collection of instantiated variables must satisfy all the constraints in C for Q' to be a refined query of Q wrt D.

3.2 Metrics for Comparing Refined Queries

Since there are generally many refined queries for a given why-not question, it is useful to have some metric to compare the quality of refined queries so that only the "good" refined queries are returned as possible explanations. There are two obvious desiderata for refined queries that can be used for this purpose.

Dissimilarity metric. First, a refined query should be as similar as possible to the original input query. This has intuitive appeal since a refined query that is minimally modified from the original query is likely to retain as much of the intention of the original input query. Moreover, by comparing the small differences between the two queries, it also serves to pinpoint to the user the "errors" she has made in her initial query. Thus, a refined query that simply modifies only one of the selection predicate is more similar to the input query than another refined query that involves a different set of relations from the original query.

Given an input query Q and a refined query Q', we compare the similarity of Q and Q' by measuring the minimum edit distance of transforming Q to Q'. Thus, two queries are more similar (or less dissimilar) if their edit distance is smaller. Since the output of Q and Q' are union-compatible (i.e., the lists of attributes in the select-clause of Q and Q' are equivalent), we only consider edit operators to transform Q to Q' in terms of modifying the query's from-clause and where-clause; the corresponding modifications to the query's select-clause and group-by-clause are trivial and are not considered in the edit distance computation. The four key edit operations considered are: (O_1) modify the constant value of a selection predicate in the where-clause, (O_2) add a selection predicate in the where-clause, (O_3) add/remove a join predicate in the where-clause, and (O_4) add/remove a relation in the from-clause. Note that there is no explicit edit operator for removing a selection predicate as this can be modeled by O_1; i.e., the removal of a selection predicate is effectively equivalent to modifying its range of selection values to cover the whole domain of the attribute. Furthermore, when O_4 is used to remove a relation R_i in the from-clause, all the selection and join predicates that are associated with R_i are also removed as part of the edit operation.

Let w_i denote the cost of the edit operation O_i, $i \in [1, 4]$. It is reasonable to assume that $w_1 < w_2 < w_3 < w_4$[2]. Let n_i denote the total number of O_i operations used in a transformation of Q to Q', $i \in [1, 4]$. The edit distance for this transformation is given by $\sum_{1 \le i \le 4}(w_i \times n_i)$. We refer to minimum edit distance to transform Q to Q' as the *dissimilarity measure* between Q and Q', which can be computed efficiently for a given Q and Q'.

Imprecision metric. Second, the refined query should be as precise as possible in terms of its result. Ideally, the result of the refined query Q' should contain only the result of the original query Q and the set of matching tuples for the why-not tuples. Any additional tuples returned in $Q'(D)$ are considered to be irrelevant tuples that should be minimized. Given a refined query Q' for a why-not question (S, C), let $R \subseteq Q'(D)$ denote a minimal set of matching tuples in $Q'(D)$ for the why-not tuples in S. The imprecision metric for Q' is defined to be the number of irrelevant tuples in $Q'(D)$ which is given by $|Q'(D) - Q(D) - R|$.

Skyline refined queries. Thus, a refined query is considered to be good if both its dissimilarity and imprecision metrics are low. Among all the possible refined queries for a why-not question, we are interested in the set of skyline refined queries defined as follows [1]. Given two different refined queries Q_1 and Q_2, we say that Q_1 *dominates* Q_2 if (1) both the metrics of Q_1 are at least as low as those of Q_2 and (2) for at least one of the metrics, Q_1's value is strictly

[2]In our experimental study, we use $w_1 = 1$, $w_2 = 3$, $w_3 = 5$ and $w_4 = 7$.

lower than that of Q_2's. We define a refined query Q' to be a *skyline refined query* (or skyline query) if Q' is not dominated by any other refined query. Thus, given a why-not question, our goal is to compute skyline refined queries to explain the question.

EXAMPLE 4. *Consider a query Q_1 on the Basketball data set that finds players who have "block" statistics no greater than 30 and "steal" statistics no greater than 150; i.e., Q_1: SELECT name FROM Player P, Regular R WHERE P.pID = R.pID and blk $\leq$ 30 AND stl $\leq$ 150. The output includes only one player "A". The why-not question $S = \{$("B")$\}$ asks why player "B" is excluded from the result.*

Consider the following refined query Q'_1: SELECT name FROM Player P, Regular R WHERE P.pID = R.pID and blk $\leq$ 35 AND stl $\leq$ 200. Observe that Q'_1 is derived from Q_1 by applying O_1 edit operations on both the selection predicates of Q_1, and the output of Q'_1 is {"A", "B", "D"}. Thus, the dissimilarity and the imprecision of Q'_1 (wrt Q_1) are $2w_1$ and 1, respectively.

Consider yet another refined query Q''_1: SELECT name FROM Player P, Regular R WHERE P.pID = R.pID and blk $\leq$ 50 AND stl $\leq$ 240. The output of Q''_1 is {"A", "B", "C", "D"}; and the dissimilarity and imprecision of Q''_1 are $2w_1$ and 2, respectively. Thus, Q'_1 dominates Q''_1, and Q'_1 is considered to be a better refined query than Q''_1. □

3.3 Explaining with ConQueR

In this section, we present an overview of our approach named ConQueR, for **Con**straint-based **Que**ry **R**efinement, to explain why-not questions by automatically generating one or more refined queries.

ConQueR is designed to be a similarity-driven approach in that it tries to generate refined queries with low dissimilarity values before considering more precise refined queries that have higher dissimilarity values. Given a why-not question (S, C) for a query Q on database D, ConQueR will first consider refined queries Q' that have the same query schema (i.e., queries with the same from-clause and join predicates) as Q. That is, ConQueR tries to derive Q' by simply modifying selection predicate(s) in Q to explain the why-not tuples while minimizing the imprecision metric. If such refined queries exist, ConQueR will only generate skyline refined queries that all share the same query schema as Q. However, if no such refined query exists, ConQueR then looks for refined queries that have a slightly different query schema (i.e., with a slightly higher dissimilarity value), and so on. Thus, ConQueR effectively iterates over a sequence of query schemas $QS_1, \cdots, QS_k$ to search for refined queries: QS_1 is the query schema of the input query Q, and schema QS_{i+1} is considered only if there are no refined queries with schema $QS_1, \cdots, QS_i$. The sequence of query schemas considered are (approximately) of increasing dissimilarity metric values, and if QS_k is the first query schema in the sequence to contain refined queries, ConQueR will generate all skyline refined queries with schema QS_k as possible explanations to the why-not question.

The architecture of ConQueR consists of two key components, ConQueR^s and ConQueR^p. Given a query Q on a database D and a why-not question (S, C) on $Q(D)$, ConQueR^s first computes refined queries Q'_s for the why-not question that are as similar as possible to Q (i.e., each Q'_s has a low dissimilarity value). Next, ConQueR^p takes each refined query Q'_s computed by ConQueR^s to derive skyline refined queries that are more precise than Q'_s by adding various additional predicates to Q'_s to improve its precision.

	pID	name	team	year	*pts*	*blk*	*stl*	*reb*
t_1	P1	A	GSW	1973	2009	30	150	40
t_2	P2	B	SEA	1994	1689	35	200	281
t_3	P2	B	SEA	1995	1563	50	240	339
t_4	P3	C	CHI	1992	2541	45	220	361
t_5	P4	D	LAL	1995	1567	30	162	359

$Q^*_\emptyset(D) = Player \bowtie_{pID} Regular$

Figure 2: Example 5

3.3.1 Notations & Definitions

Given a SQL query Q, we use $rel(Q)$ to denote the set of relations in the from-clause of Q; $proj(Q)$ to denote the set of attributes in the select-clause of Q; $sel(Q)$ to denote the set of selection predicates in the where-clause of Q; and $join(Q)$ to denote the set of join predicates in the where-clause of Q. Thus, the *query schema* of a query Q is given by $rel(Q)$ and $join(Q)$. We use ℓ to denote the number of selection predicates in Q; i.e., $|sel(Q)| = \ell$.

Consider the generation of a refined query Q' for a why-not question (S, C) on $Q(D)$ that shares the schema as Q. Conceptually, ConQueR first computes an intermediate query, denoted by $Q^*_\emptyset$, on D, where $rel(Q^*_\emptyset) = rel(Q)$, $join(Q^*_\emptyset) = join(Q)$, $sel(Q^*_\emptyset) = \emptyset$, and $proj(Q^*_\emptyset)$ consists of all the distinct attributes in $rel(Q^*_\emptyset)$. The refined query Q' is derived from $Q^*_\emptyset(D)$ as follows: $Q' = \pi_L(\sigma_P(Q^*_\emptyset))$, where $L \subseteq proj(Q^*_\emptyset)$ is a list of appropriate attributes corresponding to $proj(Q)$ so that Q and Q' are union-compatible, and P contains an appropriate set of selection predicates such that Q' is a refined query for the why-not question. Determining L from Q and $proj(Q^*_\emptyset)$ is straightforward, and the main challenge in the derivation of Q' is determining P (i.e., $sel(Q')$).

For each why-not tuple $t_i \in S$, let $M_i \subseteq Q^*_\emptyset(D)$ denote the subset of tuples in $Q^*_\emptyset(D)$ that are matching tuples of t_i. Note that for Q' to be a refined query of Q that explains all the why-not tuples, it is necessary for each M_i to be non-empty; otherwise, if some M_j is empty, then Q' will not be able to account for the why-not tuple t_j.

EXAMPLE 5. *Consider again query Q_1 in Example 4, where $Q_1(D) = \{$("A")$\}$. Consider the derivation of a refined query Q' (with the same schema as Q_1) to explain a why-not tuple $t_w =$("B"). The intermediate query $Q^*_\emptyset$ to derive Q' has $rel(Q^*_\emptyset) = \{Player, Regular\}$ and $join(Q^*_\emptyset) = \{Player.pID = Regular.pID\}$. The output of $Q^*_\emptyset$ on D is shown in Figure 2, and the set of matching tuples in $Q^*_\emptyset(D)$ for t_w is given by $M_w = \{t_2, t_3\}$. Thus, $Q'(D)$ needs to include t_2 or t_3 in order to account for the why-not tuple t_w.* □

4. EXPLAINING SPJ QUERIES

This section presents how ConQueR generates refined queries Q' to explain why-not questions (S, C) on SPJ queries Q. We first consider the simpler case where Q' and Q share the same schema: Section 4.1 explains how ConQueR^s generates refined queries with low dissimilarity values and Section 4.2 explains how ConQueR^p enhances these queries to improve their precision. Section 4.3 considers the more general case where the schema of Q and Q' are different.

For simplicity, we assume that there are no variables in the why-not tuples and therefore also no constraints on the why-not tuples (i.e., $C = \emptyset$). Details on how to process general SPJ queries are given elsewhere [14].

4.1 Modifying Selection Predicates

In this section, we explain how ConQueR^s generates refined queries Q' that have the same schema as Q. To maximize the similarity of Q' and Q, ConQueR^s derives Q' from Q by simply modifying some selection predicate(s) in Q. To simplify the presentation, we first consider the scenario where there is exactly one why-not tuple (i.e., $S = \{t_1\}$), and discuss the handling of the general scenario with multiple why-not tuples in Section 4.1.1.

For simplicity and without loss of generality, let the selection predicates in Q be of the form: $sel(Q) = \{A_1 \leq v_1, \cdots, A_\ell \leq v_\ell\}$, $\ell \geq 1$. Since Q' is derived from Q by modifying some selection predicates, let v_i' denote the modified value of v_i in Q', for $i \in [1, \ell]$.

Let Q^* denote the query that is exactly the same as Q except that $proj(Q^*)$ includes all the distinct attributes in $rel(Q)$; i.e., $Q^* = \sigma_P(Q^*_\emptyset)$ where $P = sel(Q)$. Thus, $Q^*(D)$ is the subset of tuples in $Q^*_\emptyset(D)$ that form $Q(D)$ when $Q^*(D)$ is projected on $proj(Q)$. For each selection predicate attribute A_i, $i \in [1, \ell]$, define $v_i^{max} = max_{t \in Q^*(D)}(t.A_i)$. For $Q'(D) \supseteq Q(D)$, we must have $v_i' \geq v_i^{max}$, for $i \in [1, \ell]$.

For Q' to account for the why-not tuple t_1, $Q'(D)$ must contain at least one tuple from M_1[3]. However, to minimize the imprecision of Q', $Q'(D)$ should not contain more than one tuple from M_1. Thus, each tuple in M_1 contributes to a refined query Q'. For $Q'(D)$ to contain a tuple $t_m \in M_1$, we must have $v_i' \geq t_m.A_i$, for $i \in [1, \ell]$. Therefore, combining the above two requirements, for $Q'(D)$ to contain t_m and $Q'(D) \supseteq Q(D)$, $sel(Q')$ is specified by setting $v_i' = \max\{v_i^{max}, t_m.A_i\}$, for $i \in [1, \ell]$. Note that while it is possible to generate other refined queries Q'' that also satisfy the two requirements by setting some $v_j' > \max\{v_j^{max}, t_m.A_j\}$, the imprecision of Q'' will be at least as high as that of Q' which means that Q'' will be dominated by Q'. Therefore, to generate only skyline refined queries, we must have $v_i' = \max\{v_i^{max}, t_m.A_i\}$, for $i \in [1, \ell]$.

In addition, since we are interested only in skyline refined queries, the number of refined queries considered can be reduced by considering only the "skyline" tuples in M_1. Consider two tuples $t_x, t_y \in M_1$, and let Q'_x and Q'_y denote the refined queries corresponding to t_x and t_y, respectively. We say that t_x dominates t_y if (1) $t_x.A_i \leq t_y.A_i$ for $i \in [1, \ell]$, and (2) at least one of the inequalities in (1) is strict. The skyline tuples in M_1 are the tuples that are not dominated by any tuple in M_1. If t_x dominates t_y, it follows that Q'_x dominates Q'_y. Thus, to generate skyline refined queries, we only need to consider the skyline tuples in M_1.

EXAMPLE 6. *Reconsider Example 4 where the input query is Q_1 and the why-not tuple is t =("B"). Let A_1 and A_2 denote the two selection attributes blk and stl, respectively. We have $Q^* = \sigma_{blk \leq 30 \wedge stl \leq 150}(Q^*_\emptyset)$. Thus, $Q^*(D) = \{t_1\}$, $v_1^{max} = 30$, and $v_2^{max} = 150$. Since $M_1 = \{t_2, t_3\}$, there are two possible refined queries corresponding to these matching tuples for t. To generate the refined query Q'_1 such that $Q'_1(D)$ contains $t_2 \in M_1$, ConQueR^s modifies the two predicates in $sel(Q)$ into $blk \leq 35$ and $stl \leq 200$, and obtains the refined query Q'_1 as shown in Example 4.*

Similarly, to generate the refined query Q''_1 such that $Q''_1(D)$ contains $t_3 \in M_1$, ConQueR^s modifies the two predicates in $sel(Q)$ into $blk \leq 50$ and $stl \leq 240$, and obtains the refined query Q''_1 as given in Example 4.

However, by considering only the skyline tuples in M_1, ConQueR^s actually would not have considered Q''_1 since t_3 is dominated by t_2 which means that Q''_1 is not a skyline refined query. □

Finally, to generate the skyline refined queries from the set of queries corresponding to the skyline tuples in M_1, ConQueR^s needs to compute and compare the imprecision values of these queries by computing their results.

4.1.1 Handling multiple why-not tuples

The above technique can be easily extended to handle the general case where there are multiple why-not tuples; i.e., $S = \{t_1, \cdots, t_n\}$, $n > 1$. Specifically, for each M_i, $i \in [1, n]$, ConQueR^s first computes the set of skyline tuples, denoted by SL_i, in M_i. Next, ConQueR^s enumerates different refined queries Q' that correspond to different subsets $M' \subseteq \cup_{i=1}^n SL_i$ of matching tuples, where each M' consists of one tuple from each of SL_i, $i \in [1, n]$. For example, if t'_j is the tuple selected from each SL_j, $j \in [1, n]$, then the selection condition in the refined query Q' is specified by setting $v_i' = \max\{v_i^{max}, t'_1.A_i, \cdots, t'_m.A_i\}$, $i \in [1, \ell]$.

4.2 Improving Precision with More Predicates

Since the refined queries Q' produced by ConQueR^s are generated by simply modifying the selection predicates in Q, there are likely to be many irrelevant tuples in $Q'(D)$. In this section, we explain how ConQueR^p improves the precision of the refined queries Q' produced by ConQueR^s by adding additional selection predicates to Q' to reduce the irrelevant tuples in $Q'(D)$ while ensuring that the enhanced query Q' remains a refined query for the input why-not question. Thus, the refined queries produced by ConQueR^p tradeoffs low dissimilarity values for low imprecision values.

Consider a refined query Q' produced by ConQueR^s that corresponds to the subset of matching tuples $T \subseteq \bigcup_{i \in [1,n]} M_i$ to explain the set of why-not tuples $S = \{t_1, \cdots, t_n\}$. Let $\mathcal{A}$ denote the set of attributes in $rel(Q')$ that do not have a selection predicate in $sel(Q')$. For each attribute $A_i \in \mathcal{A}$, ConQueR^p can add the following predicate to try to reduce the irreverent tuples in $Q'(D)$: "$A_i \leq max_{t \in Q^*(D) \cup T}(t.A_i)$"

Thus, there are a total of $|\mathcal{A}|$ possible additional predicates that ConQueR^p can introduce into Q' to reduce its imprecision. As the problem to maximize the elimination of irrelevant tuples using the minimum number of additional predicates is NP-hard (shown by reduction from set-covering problem), ConQueR^p uses a standard greedy heuristic to select the additional selection predicates by choosing the predicates in non-increasing order of the number of irrelevant tuples that they can eliminate.

4.3 Refined Queries with Different Schema

When ConQueR is unable to find refined queries having the same query schema as Q, ConQueR will consider other similar

[3]Note that since $proj(Q') \subseteq proj(Q^*_\emptyset)$ and $M_i \subseteq Q^*_\emptyset(D)$, when we say that $Q'(D)$ must "contain" one tuple t from M_i, what we mean is that $Q'(D)$ must contain one tuple t that is a projection of some tuple t_{int} from M_i; i.e., $t = \pi_L(t_{int})$, where $L = proj(Q')$.

schemas, roughly in increasing order of their dissimilarity metrics. In this section, we explain how ConQueR enumerates alternative query schemas and generates refined queries for such schemas.

Enumerating schemas. ConQueR uses a simple heuristic to enumerate query schemas approximately in increasing order of dissimilarity metrics. Let S_R denote the set of the relations in $rel(Q)$ that contain the attributes in $proj(Q)$. These relations need to be retained in Q' so that the $proj(Q')$ and $proj(Q)$ are equivalent. Thus, ConQueR generates a different schema by adding some relation(s), removing some relation(s) from $rel(Q) - S_R$, or adding/removing some join predicates. Since the most costly edit operation is add/remove relations, ConQueR will consider schemas in increasing order of the total number of relations added/removed. For each new relation R added, ConQueR enumerates different ways to connect R to the existing relations via adding new join predicates. A candidate query schema that contains more than one connected component of relations will be ignored by ConQueR.

Generating refined queries. Consider the general case where refined queries Q' are to be generated for a schema that is different from that of Q involving a set of relations R and a set of join predicates J. ConQueR first rewrites the input why-not question (S, C) into an equivalent question as follows: by assuming that $Q(D)$ is empty, ConQueR transforms the why-not question into (S', C), where $S' = Q(D) \cup S$. The transformed why-not question can be processed using the previously discussed techniques as follows. First, ConQueR^s generates a refined query Q' with low dissimilarity value such that $rel(Q') = R$, $join(Q') = J$, $sel(Q') = \emptyset$, and $proj(Q')$ contains the corresponding attributes in $proj(Q)$. Note that if $Q'(D)$ can not account for all the why-not tuples in S', then there are no refined queries for this schema and ConQueR will consider another query schema for possible refined queries. If Q' is a refined query, ConQueR^p will try to enhance the precision of Q' by adding additional selection predicates. Specifically, ConQueR^p first computes the set of skyline tuples SL_i in M_i (wrt all attributes in $rel(Q')$) for each why-not tuple $t_i \in S'$ based on the techniques in Section 4.1.1, and then uses the techniques described in Section 4.2 to enhance $sel(Q')$.

EXAMPLE 7. *Consider again query Q_1 in Example 4 and another why-not question $S = \{t_w\}$, where t_w = ("E"). Here,* ConQueR *is unable to derive any refined query with the same schema as Q_1 because t_w does not have any matching tuples in $Q^*_\emptyset(D)$. To generate refined queries with a different schema from Q_1,* ConQueR *transforms the why-not question to become $S' = \{$("A"), ("E")$\}$ and is now able to derive a refined query Q'_3 that involves the join between Player and Playoff: SELECT name FROM Player, Playoff WHERE Player.pID = Playoff.pID AND pts $\leq$ 2029.* □

In the event that ConQueR cannot find any SPJ refined queries, ConQueR will resort to derive SPJU refined queries Q' of the form: $Q' = Q$ *union* Q_s, such that Q_s accounts for the why-not tuples in S. To derive Q_s, ConQueR first needs to determine $rel(Q_s)$. Since the why-not tuples in S are essentially contained in a $|proj(Q)|$-column table T, $rel(Q_s)$ must be selected such that for each column C_i in T, there must be a "matching attribute" A'_i in some relation in $rel(Q_s)$ such that the set of constant values in C_i are contained by the values in A'_i. For each potential candidate for $rel(Q_s)$, Q_s is constructed by ConQueR^s as follows: $sel(Q_s)$ is defined to be an empty set, $join(Q_s)$ is defined to be the set of foreign-key join predicates among the relations in $rel(Q_s)$, and $proj(Q_s)$ is defined to be set of matching attributes. If the resultant query schema (defined by $rel(Q_s)$ and $join(Q_s)$) is not a single connected component, then the candidate schema for Q_s is ignored. Otherwise, if $Q_s(D)$ can account for all the why-not tuples, then the query Q_s produced by ConQueR^s can be further enhanced by ConQueR^p to improve its precision.

5. EXPLAINING SPJA QUERIES

In this section, we explain how ConQueR generates refined queries for SPJA queries. For simplicity and without loss of generality, we assume there is only a single aggregated attribute in $proj(Q)$ based on SUM operator, and we use A_a to denote the attribute in $proj(Q)$ being aggregated and use A_{agg} to denote $SUM(A_a)$. We also assume that the domain of A_a contains positive values. Details on how the ideas can be generalized for other cases are given elsewhere [14].

As the examples in the introduction illustrated, ConQueR can handle two types of why-not questions on SPJA queries. In the first basic type of why-not questions, each why-not tuple corresponds to either some existing tuple $t_i \in Q(D)$ or some missing tuple t_i, and the question asks why $t_i.A_{agg}$ is not greater than some value K_i. In the second more complex type of why-not questions, it involves at least two why-not tuples, t_1 and t_2 (which may be existing or missing tuples), and the explanation sought is to clarify on the relationship between their A_{agg} attribute values. For example, if t_1 and t_2 are two existing tuples in $Q(D)$ with $t_1.A_{agg} \leq t_2.A_{agg}$, then the why-not question asks why $t_1.A_{agg}$ is not greater than $t_2.A_{agg}$.

To simplify the presentation, we shall assume that for each why-not tuple t in S, the components corresponding to the non-aggregated values (i.e., group-by attributes) in t all have constant values. The details for handling the more general scenario where some non-aggregated components are don't-care values or variables are given elsewhere [14].

Whereas the processing of why-not questions on SPJ queries requires $Q'(D)$ to contain a single matching tuple from M_i for each why-not tuple $t_i \in S$, the processing for SPJA queries is more complex as $Q'(D)$ needs to contain a subset of matching tuples from M_i to satisfy the aggregation constraint of each why-not tuple $t_i \in S$.

5.1 Basic Why-not Questions

Let us consider the case where Q and Q' have the same query schema, and there is exactly one why-not tuple $S = \{t_1\}$ which is a missing tuple (i.e., $t_1 \notin Q(D)$) and the constraint in C requires $t_1.A_{agg} > K$.

As in Section 4.1, we assume that $sel(Q) = \{A_1 \leq v_1, \cdots, A_\ell \leq v_\ell\}$, $\ell \geq 1$. Let v'_i denote the modified value of v_i in Q', for $i \in [1, \ell]$. The definitions of $Q^*(D)$ and v_i^{max} in Section 4.1 are used here as well. Let J_q denote the subset of tuples in $Q^*_\emptyset(D)$ that are the matching tuples of $Q(D)$; i.e., for every tuple $t_q \in J_q$, there exists one tuple $t \in Q(D)$ such that for every non-aggregated attribute component of t_q, the corresponding component of t has the same value.

Naive ConQueR (ConQueR$^-$). To motivate the optimizations adopted by ConQueR to process why-not questions

	pID	name	team	year	*pts*	*blk*	*stl*	*reb*
t_1	P1	A	GSW	1973	2029	40	100	30
t_2	P2	B	SEA	1994	3000	65	150	181
t_3	P2	B	CHI	1995	2200	50	120	161
t_4	P2	B	LAL	1996	2500	70	110	200
t_5	P4	D	LAL	1995	2300	70	150	150
t_6	P5	E	DEN	2000	1689	35	200	381

$Q^*_\emptyset(D) = Player \bowtie_{pID} Playoff$

Figure 3: Example 8

on SPJA queries, we first present a simpler variant of ConQueR, denoted by ConQueR$^-$.

For each selection predicate attribute A_i, $i \in [1, \ell]$, let lb_i denote the smallest A_i value among $\{t.A_i \mid t \in M_1\}$ that satisfies the constraint $\sum_{t\in M_1, t.A_i<lb_i}(t.A_a) \leq K < \sum_{t\in M_1, t.A_i\leq lb_i}(t.A_a)$. It follows that for Q' to be a refined query for the why-not question, we must have $v'_i \geq lb_i$, for $i \in [1, \ell]$. Moreover, for $Q'(D) \supseteq Q(D)$, we must have $v'_i \geq v_i^{max}$, for $i \in [1, \ell]$ as explained in Section 4.1.

Thus, based on the above two constraints, ConQueR$^-$ enumerates all potential values for each $v'_i \in V_i$, where $V_i = \{t.A_i \mid t \in M_1 \wedge t.A_i \geq \max\{lb_i, v_i^{max}\}\}$. Each combination $(v'_1, \cdots, v'_\ell)$ considered corresponds to a potential refined query Q'. Therefore, if (1) $Q'(D)$ can account for all the why-not tuples in S and (2) $Q'(D) \supseteq Q(D)$, then Q' is a refined query for the why-not question. Note that for $Q'(D) \supseteq Q(D)$, it is necessary that $Q'(D)$ does not contain any tuples in $J_q - Q^*(D)$[4].

Even with the use of constraints, the total number of potential refined queries to be considered, given by $\prod_{i=1}^{\ell} |V_i|$, is rather large. For efficiency reason, ConQueR$^-$ adopts a two-step approach to generate refined queries. In the first step, a heuristic is used to choose a subset A' of selection attributes in $sel(Q)$. In the second step, A' is used to generate the potential refined queries. Thus, the number of refined queries considered is reduced to $\prod_{A_j\in A'} |V_j|$. While this approach improves efficiency, the tradeoff is that the refined queries generated have higher dissimilarity values since not all the selection attributes in $sel(Q)$ appear in Q'. In ConQueR$^-$, the heuristic for selecting A' uses an input control parameter θ[5] so that $\prod_{A_j\in A'} |V_j|$ is no larger than θ. To minimize the dissimilarity values of the refined queries, ConQueR$^-$ uses a simple greedy heuristic to maximize the number of selected attributes in A' by selecting the attributes A_j in non-descending order of $|V_j|$.

Example 8. *Consider a query Q_2 on the Basketball data set that finds players and their total points scored in play-off games that satisfy some conditions on their block and steal statistics: SELECT name, SUM(pts) FROM Player, Playoff WHERE Player.pID = Playoff.pID and blk $\leq$ 40 AND stl $\leq$ 100 GROUP BY name. The output contains only one tuple ("A", 2029). Consider the why-not question $S = \{t_w\}$ with t_w = ("B", \$x) and $C = \{\$x > 3500\}$ which asks why "B", with a total score of greater than 3500, is missing from the output.*

[4]Suppose Q' selected a tuple $t_q \in J_q - Q^*(D)$, and let $t_d \in Q(D)$ be the tuple in $Q(D)$ that corresponds to t_q. Then $t_d.A_{agg}$ in $Q'(D)$ will be greater than $t_d.A_{agg}$ in $Q(D)$; i.e., $Q'(D) \not\supseteq Q(D)$.

[5]In our experiments, we set $\theta = 100000$.

ConQueR$^-$ *is able to derive refined queries Q' that have the same schema as Q_2 for this why-not question. The output of the intermediate query $Q^*_\emptyset$ to derive Q' is shown in Figure 3. Let A_1 and A_2 denote the two selection predicates blk and stl, respectively. We have $Q^* = \sigma_{blk\leq 40 \wedge stl\leq 100}(Q^*_\emptyset)$. Thus, $Q^*(D) = \{t_1\}$, $v_1^{max} = 40$, and $v_2^{max} = 100$. The set of matching tuples in $Q^*_\emptyset(D)$ for t_w is given by $M_w = \{t_2, t_3, t_4\}$.* ConQueR$^-$ *derives $lb_1 = 65$ and $lb_2 = 120$; therefore, $V_1 = \{65, 70\}$ and $V_2 = \{120, 150\}$.*

ConQueR$^-$ *generates four candidate refined queries as follows. First,* ConQueR$^-$ *selects the set of attributes $A' = \{A_1, A_2\}$ to be used for the refined queries. Next, based on V_1 and V_2, a candidate refined query is generated corresponding to each of the four combinations of (v'_1, v'_2), where $v'_1 \in V_1$ and $v'_2 \in V_2$. Among these four candidates, the query Q'_2 corresponding to the combination $(65, 120)$, given by: SELECT name, SUM(pts) FROM Player, Playoff WHERE Player.pID = Playoff.pID and blk $\leq$ 65 AND stl $\leq$ 120 GROUP BY name, is not a valid refined query. This is because the output of Q'_2, which contains the tuples ("A", 2029) and ("B", 2200), does not account for the why-not tuple t_w. The candidates corresponding to the remaining three combinations are valid refined queries.* □

Optimization. In this section, we present the optimizations adopted by ConQueR to optimize the generation of refined queries. ConQueR is also based on the two-step approach as ConQueR$^-$, where it first selects a subset of attributes A' followed by using A' to generate potential refined queries. However, ConQueR exploits additional properties to prune away the useless candidate refined queries. Thus, ConQueR is able to more efficiently generate the same set of refined queries as ConQueR$^-$.

Let $A' = \{A_1, \cdots, A_m\}$ denote the set of attributes selected by the greedy heuristic in the first step, where $|V_1| \leq \cdots \leq |V_m|$. Let $M_1 = \{x_1, x_2, \cdots, x_n\}$, where $x_1.A_1 \leq x_2.A_1 \leq \cdots \leq x_n.A_1$. Let x_s be the "first" tuple in M_1 such that $\sum_{t\in M_1, t.A_1\leq x_s.A_1} t.A_a > K$ and $\sum_{t\in M_1, t.A_1\leq x_{s-1}.A_1} t.A_a \leq K$. Observe that for Q' to be a refined query, $Q'(D)$ must contain at least one matching tuple from $\{x_s, \cdots, x_n\}$; if not, the selected matching tuples will not be able to account for the missing why-not tuple t_1. Based on this observation, we can view the collection of candidate refined queries as being partitioned into $(n - s + 1)$ groups $G_s, G_{s+1}, \cdots, G_n$ such that for each refined query Q' in group G_i, the matching tuples in M_1 that are selected by Q' include x_i and a (possibly empty) subset of $\{x_1, \cdots, x_{i-1}\}$.

Thus, ConQueR enumerates the candidate refined queries in $(n - s + 1)$ iterations, where at the j^{th} iteration for $j \in [1, n - s + 1]$, Q' selects the matching tuples from M_1 that contains x_{s+j-1} and a subset of $\{x_1, \cdots, x_{s+j-2}\}$. More specifically, in the j^{th} iteration, $j \in [1, n-s+1]$, the following values of $v'_i, i \in [1, m]$ are being considered:

1. v'_1 is set to $\max\{v_1^{max}, x_{s+j-1}.A_1\}$ to ensure that x_{s+j-1} is selected from M_1 and that $Q'(D) \supseteq Q(D)$.

2. For each $v'_i, i \in [2, m]$, the values considered for v'_i are selected from the set $S_i = \{x_1.A_i, \cdots, x_{s+j-1}.A_i\}$ and must satisfy the following constraints:

   (a) $v'_i \geq lb_i$ to ensure that Q' is a refined query;

   (b) $v'_i \geq v_i^{max}$ to ensure that $Q'(D) \supseteq Q(D)$; and

(c) $v'_i \geq x_{s+j-1}.A_i$ to ensure that x_{s+j-1} is selected by Q'.

Thus, each combination $(v'_1, \cdots, v'_m)$ considered corresponds to a candidate refined query Q'. The total number of combinations considered by ConQueR is $\sum_{i=1}^{n}(i^{m-1})$ in the worst case. Our experimental results in Section 7 showed that the pruning optimization enables ConQueR to be 2 to 10 times faster than ConQueR$^-$.

EXAMPLE 9. *This example reconsiders query Q_2 in Example 8 to illustrate how the above optimizations enable* ConQueR *to prune away the invalid candidate refined query generated by* ConQueR$^-$. ConQueR *first derives $M_1 = \{x_1, x_2, x_3\}$, where $x_1 = t_3$, $x_2 = t_2$ and $x_3 = t_4$ such that $x_1.A_1 \leq x_2.A_1 \leq x_3.A_1$. The "smallest" tuple x_s that satisfies the aggregation constraint is x_2.* ConQueR *enumerates the candidate refined queries in two iterations as follows. In the first iteration, v'_1 is set to 65 and v'_2 is selected from the set $S_2 = \{120, 150\}$ which results in v'_2 being set to $v'_2 \geq 150$ (i.e., $v'_2 = 150$). In the second iteration, v'_1 is set to 70 and v'_2 is selected from the set $S_2 = \{110, 120, 150\}$ which results in v'_2 being set to $v'_2 \geq 120$ (i.e., $v'_2 \in \{120, 150\}$). Thus,* ConQueR *generates only the candidate queries corresponding to the combinations $(65, 150)$, $(70, 120)$, and $(70, 150)$, which is a proper subset of those generated by* ConQueR$^-$. □

5.2 Complex Why-not Questions

The techniques presented in the previous section to process basic why-not questions on SPJA queries can be extended to handle the more complex why-not questions as well. Consider a complex why-not question on SPJA queries with $S = \{t_1, \cdots, t_k\}$ and the constraint in C requires that $t_1.A_{agg} < \cdots < t_k.A_{agg}$.

The approach for enumerating candidate refined queries in this case follows the same approach discussed in the previous section except that each V_i is now defined as follows: $V_i = \{t.A_i \mid t \in \mathcal{P} \wedge t.A_i \geq v_i^{max}\}$, where $\mathcal{P} = M_1 \cup \cdots \cup M_k$.

6. ALTERNATIVE APPROACH: TALOS$^+$

In this section, we present an overview of an alternative approach to generate refined queries for SPJ/SPJA queries that is based on extending a recent technique called TALOS [15] which is designed to derive *instance-equivalent* queries for an input SPJ query on a database.

Given a query Q on a database D, the goal of TALOS is to generate query-based characterizations of the query result $Q(D)$ by deriving instance-equivalent queries (IEQs) Q'. Two queries Q and Q' are defined to be instance-equivalent wrt a database D if their results on D are equivalent; i.e., $Q(D) = Q'(D)$. TALOS generates instance-equivalent queries Q' for Q on D by considering various query schema for Q' based on the $proj(Q)$ and $join(Q)$. For each candidate schema, TALOS can easily determine $rel(Q')$, $join(Q')$, and $proj(Q')$. In contrast to ConQueR which uses a constraint-based approach to derive $sel(Q')$, TALOS uses a classification-based approach to determine $sel(Q')$ by constructing decision trees. By enumerating different decision trees to generate different sets of selection predicates for $sel(Q')$, different IEQs Q' are derived for Q.

We briefly overview the data classification approach in TALOS. First, TALOS computes a relation J by joining all relations in $rel(Q')$ using $join(Q')$. TALOS then builds some decision trees DT to classify tuples corresponding to $Q(D)$ from all tuples of J by using the attributes of J as the splitting attributes. After a decision tree DT has been built, TALOS derives Q' by classifying a leaf node as *positive* if the ratio of the number of its negative tuples to the number of its positive tuples is smaller than some threshold value. Each internal node in DT corresponds to a selection predicate on some attribute of J, and each root-to-positive-leaf path in DT corresponds a conjunctive predicate C_j on J. Thus, each decision tree DT yields the selection predicate of Q' of the form C_1 *or* C_2 $\cdots$ *or* C_ℓ. After producing one decision tree, TALOS builds other decision trees by considering the attributes of J that do not appear in the selection clauses of the derived IEQ Q' as the splitting attributes used in the subsequent decision tree construction process.

We have extended TALOS to generate refined queries to explain why-not questions. We refer to this extended approach as TALOS$^+$. The basic idea is to treat $Q(D)$ together with the why-not tuples as the output result of some query Q' and apply TALOS to derive IEQs for Q'. A key challenge in extending TALOS, which is a precision-oriented approach, to TALOS$^+$ is the modification of the data classification step to construct "linear" decision trees so that the refined queries generated are more similar to the input queries. Further details of TALOS$^+$ are given elsewhere [14].

	Query	Size
Q_1	$\pi_{name}\sigma_{year\geq 2000 \wedge pts>2300}$ $(Player \bowtie Regular)$	7
Q_2	$\pi_{name,team}\sigma_{year>2000 \wedge stl>50 \wedge o\_pts\geq 5000}$ $(Player \bowtie Playoff \bowtie TeamSeason)$	1
Q_3	$\pi_{name,AVG(pts)}\mathcal{G}_{name}\sigma_{year\leq 1970 \wedge pts>2600}$ $(Player \bowtie Regular)$	3
Q_4	$\pi_{name,SUM(pts)}\mathcal{G}_{name}\sigma_{year>2000 \wedge pts>2300}$ $\sigma_{blk>70}$ $(Player \bowtie Regular)$	2
Q_5	$\pi_{team,SUM(won)}\mathcal{G}_{team}\sigma_{lost<30 \wedge dpts>8000}$ $\sigma_{year\geq 2008}$ $(Team \bowtie TeamSeason)$	2
Q_6	$\pi_{supplier.name}\sigma_{acctbal>4000 \wedge regionkey<3}$ $(supplier \bowtie nation)$	3284
Q_7	$\pi_{customer.name}\sigma_{acctbal>9900}$ $(customer \bowtie nation)$	1380
Q_8	$\pi_{part.name}\sigma_{retailprice>800 \wedge supplycost>990}$ $(part \bowtie part\_supp)$	7866

Table 1: Test queries for experiments

7. EXPERIMENTAL STUDY

In this section, we evaluate the effectiveness and efficiency of our proposed approach to find explanations for why-not questions. In the first set of experiments (Section 7.1), we compare the performance of our constraint-based approach, ConQueR, against the classification-based approach, TALOS$^+$, in terms of the processing efficiency as well as the quality of the derived refined queries. We also validate the efficiency of the pruning optimization in ConQueR by comparing against ConQueR$^-$. In the second set of experiments (Section 7.2), we compare the effectiveness of our query-refinement based approach to explain why-not questions against the two existing approaches [2, 7].

We used two data sets for the experiments: the NBA Basketball statistics and TPC-H data set (with a database size of 1GB). In the Basketball data set, the number of tuples in the relations *Player*, *Regular*, *Playoff*, *Team*, and *TeamSeason* are, respectively, 3863, 21376, 8347, 100 and 1307. In the TPC-H data set, the number of tuples in relations *part_supp*, *part*, *customer*, *supplier*, and *nation* are 800000, 200000, 150000, 10000, and 25, respectively. The five test

	Why-not questions
W_1	$S = \{$(Rick Barry), (Wilt Chamberlain)$\}$
W_2	$S = \{$(Michael Jordan, WAS)$\}$
W_3	$S = \{$(Kareem Abdul-Jabbar, \$$x$)$\}$, $C = \{\$x > 2000\}$
W_4	$S = \{$(Dwyane Wade,\$$x$), (LeBron James, \$$y$)$\}$, $C = \{\$x < \$y\}$
W_5	$S = \{$(CHI,\$$x$), (DEN, \$$y$), (LAL, \$$z$)$\}$, $C = \{\$x < \$y < \$z\}$
W_6	$S = \{$(Supplier1336), (Supplier9819)$\}$
W_7	$S = \{$(Customer100), (Customer197), (Customer219), (Customer468), (Customer518), (Customer780), (Customer987), (Customer1042), (Customer1370), (Customer1573)$\}$
W_8	$S = \{$(beige steel)$\}$

Table 2: Why-not questions

Query	ConQueR^s		ConQueR		TALOS$^+$	
	d	i	d	i	d	i
Q_1	2	24	14	6	17	1
Q_2	47	9562	74	0	59	0
Q_3	1	0	1	0	2	0
Q_4	3	0	3	0	12	0
Q_5	3	8	9	0	12	0
Q_6	2	2197	2	2197	2	2197
Q_7	1	1296	1	1296	1	1296
Q_8	2	762	2	762	2	762

Table 3: The dissimilarity (d) and the imprecision (i) values of refined queries

queries (Q_1-Q_5) for the Basketball data set and three test queries (Q_6-Q_8) for the TPC-H data set are shown in Table 1, where the third column indicates the number of tuples in the output of each test query. Table 2 shows the why-not questions used for these queries, where W_i denote the why-not question on query Q_i, $i \in [1, 8]$.

We used MySQL Server 5.0.51 for our database system, and all algorithms were coded using C++ and compiled with gcc. Our experiments were conducted on a dual-core, 2.33GHz PC running Linux with 3.25GB of RAM and a 200GB hard disk.

7.1 Comparing ConQueR & TALOS$^+$

In this section, we compare the performance of ConQueR and TALOS$^+$. We also included the performance of ConQueR^s to understand the tradeoffs between the two key components of ConQueR.

For both ConQueR and TALOS$^+$, we limit the maximum number of selection predicates in refined queries to be 3 times the number of selection predicates in the input query. The time taken to process each why-not question is measured as follows. For ConQueR^s, the time reported refers to the processing time to derive all the refined skyline queries; For ConQueR, the time reported is a sum of two components: (1) the time incurred by ConQueR^s to generate a set of refined skyline queries, and (2) the time taken by ConQueR^p to maximize the precision of each refined query produced by ConQueR^s and output the final skyline refined queries. For TALOS$^+$, the time reported refers to the processing time to generate only the first skyline refined query (i.e., the query corresponding to the first constructed decision tree). Note that if we had measured the total time for TALOS$^+$ to generate all skyline refined queries, the time reported for TALOS$^+$ would have been higher by a factor of 4 to 7 times. The

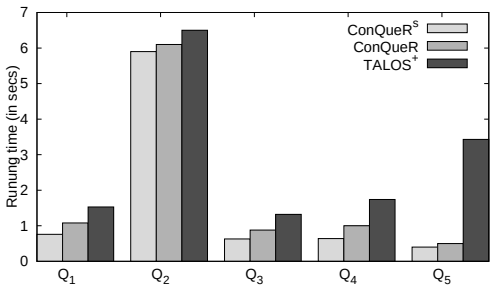

Figure 4: Running time comparisons on Basketball

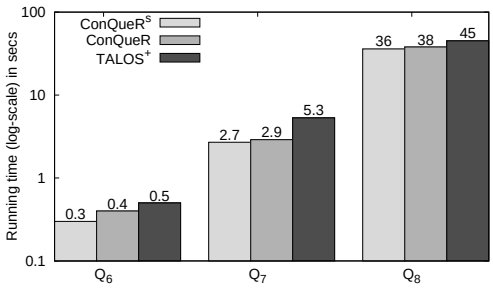

Figure 5: Running time comparisons on TPCH

quality of the refined queries are compared in terms of the dissimilarity and the imprecision metrics, where smaller values indicate better quality.

Quality of Refined Queries. Table 3 compares the quality of the refined queries. Observe that the refined queries computed by ConQueR^s have the lowest dissimilarity values but the highest imprecision values. At the other extreme, the refined queries generated by TALOS$^+$ have the lowest imprecision values but highest dissimilarity values. In contrast, the refined queries produced by ConQueR are quite similar to the original queries but also (nearly) as precise as these generated by TALOS$^+$. For TALOS$^+$, the reason for the high dissimilarity values for its refined queries is because the refined queries can include many selection attributes that are not in the original queries. ConQueR, on the other hand, first uses ConQueR^s to derive refined queries with low dissimilarity values and then enhances their precision with additional selection predicates. The overall quality of the refined queries generated by ConQueR are therefore good enough in terms of both the dissimilarity and imprecision metrics. For some queries (e.g., Q_1, Q_4), although the number of the selection attributes in the refined queries generated by ConQueR and TALOS$^+$ are nearly the same, the refined queries computed by ConQueR are relatively more similar to the input queries because ConQueR uses more attributes that appear in the original queries than TALOS$^+$.

Processing Efficiency. The running time performance comparisons are shown in Figures 4 and 5, respectively for the Basketball and TPC-H data sets. Since ConQueR^s is only one component of ConQueR, the performance of ConQueR^s is, not surprisingly, better than that of ConQueR. The experimental results show that ConQueR outperforms TALOS$^+$

Query	#candidate queries		Running time (s)	
	ConQueR	ConQueR$^-$	ConQueR	ConQueR$^-$
Q_3	18	380	0.9	1.1
Q_4	31	600	1.0	2.0
Q_5	1263	63455	0.5	53.5

Table 4: Comparison of ConQueR and ConQueR$^-$

by a factor of 1.5 to 6 times, indicating the efficiency of the constraint-based approach over the classification-based approach. The classification-based approach incurs a high computation overhead to determine optimal node splits.

Effectiveness of Pruning Optimization. To validate the effectiveness of the pruning optimization in ConQueR for processing SPJA queries, we also compare the performance of ConQueR against ConQueR$^-$.

Table 4 compares the number of considered candidate refined queries and the running times of ConQueR and ConQueR$^-$ for the SPJA queries Q_3, Q_4, and Q_5. The results clearly demonstrate the effectiveness of the pruning optimization. For queries Q_3 and Q_4, ConQueR is 1.5 to 2 times faster than ConQueR$^-$, while for query Q_5, ConQueR is two orders of magnitude faster than ConQueR$^-$. This huge performance difference is due to the significant pruning of useless candidate refined queries: the number of candidate refined queries considered by ConQueR and ConQueR$^-$ are, 1263 and 63455, respectively.

7.2 Comparison of Explanation Models

In this section, we evaluate the usefulness of our proposed query refinement approach to explain why-not questions. We also compare the explanations obtained from the two existing approaches: the approach that is based on identifying the culprit operators that filtered out the missing tuples [2], which we denote by CO, and the approach that is based on database modifications to produce the missing tuples [7], which we denote by DM.

We used the test queries on the Basketball data set (i.e, Q_1 to Q_5 in Table 1) and their corresponding why-not questions (i.e., W_1 to W_5 in Table 2). Table 5 shows the refined queries, denoted by R_i^M, computed for the why-not question W_i on query Q_i using approach M, where $M \in \{conquer^s, conquer, talos^+\}$ and the number of refined queries returned by each method. When ConQueR or ConQueR^s returns more than one refined query, we only report the one that is the most similar to the input query.

Query Q_1 finds the recent high-scoring NBA players. Although some expected well-known superstar players such as "LeBron James" and "Kobe Bryant" are included in the result, other superstar players such as "Rick Barry" and "Wilt Chamberlain" are missing from the result. The why-not question W_1 seeks an explanation for these two missing players. The CO approach would have simply identified the selection predicate "$year \geq 2000$" as the reason for Q_1 to have excluded the missing tuples. The DM approach would have suggested several possible ways to modify the data set for the two why-not tuples to be selected by Q_1. For instance, if all the attributes of $sel(Q_1)$ were allowed to be modified, then there will be a total of 224 ways to modify the existing tuples: there are 12 ways to modify the tuples in *Regular* relation to include Barry and there are 12 ways to modify the tuples in *Regular* relation to include Chamberlain. The refined query computed by ConQueR^s, however, not only implicitly points out that missing tuples are excluded due to the predicate on the *year* attribute, but it also reveals the additional information that the missing players are actually superstars in the 1960's. The refined query computed by ConQueR has higher precision as it further introduces additional selection predicates on attributes such as *weight*, *asts*, etc. The refined query computed by TALOS$^+$ has higher precision but lower similarity compared to the refined query computed by ConQueR. In fact, for the other test queries, although the refined queries computed by TALOS$^+$ have slightly higher precision (relative to those computed by ConQueR), the refined queries are very different from the original queries; for instance, the refined query computed by TALOS$^+$ for Q_4 uses a very different set of attributes from the attributes in $sel(Q_4)$.

Query Q_2 finds the players and the teams that they were playing for when the teams gained a large number of offense points and steal statistics. The why-not question W_2 asks why "Michael Jordan" and his team "WAS" is missing in the result. The CO approach would not be able to generate any explanation for this query because when Jordan was playing for WAS, he did not participate in any playoff games; thus, the why-not tuple does not have any matching tuples in the join result of *Player*, *Playoff* and *TeamSeason*. For the DM approach, if the projected attributes were not allowed to be modified, then no explanation can be given for the same reason; otherwise, there are a total of 13 ways to modify the *team* attribute of the tuples corresponding to Jordan to return the why-not tuple. Our query refinement approach, which can derive refined queries that have different schema from the input query, is able to compute a refined query that involves the relations *Player*, *Regular* and *TeamSeason*. From this refined query, the user can figure out why ("Jordan", "WAS") was missing from the original query's result: it is due to the fact Jordan participated in only regular-season games when he was playing for WAS.

For queries Q_3, Q_4 and Q_5, which are SPJA queries with complex why-not questions, the approach CO is not applicable. The approach DM, which is the most flexible approach, in general has many possible options to modify values in the data set to satisfy the aggregation constraints for these complex why-not questions. In the rest of this section, we will just focus on the explanations computed by ConQueR.

Query Q_3 computes the average of the "high points" (defined to be more than 2600 points) scored by players in regular-season games for the period until 1970. The output includes ("Rick Barry", 2775), ("Wilt Chamberlain", 3159) and ("Elgin Baylor", 2719). The why-not question W_3 asks why "Kareem Abdul-Jabbar" with an average high-point score of more than 2000 is missing from the result. The refined query computed by ConQueR indicates that the missing tuple will be included if the predicate on *pts* is modified to become "$pts \geq 2596$". This refined query turns out to be a precise refined query that returns exactly one additional tuple that matches the missing tuple.

Query Q_4 computes the total points scored by players for regular-season games that satisfy the following three conditions: $year > 2000$, $pts > 2300$, and $blk > 70$. The result contains only two tuples: ("Dwyane Wade", 2386) and ("LeBron James", 2304). The why-not question W_4 asks why the total points of James is not higher than that of Wade. The refined query computed by ConQueR modifies the three

	Refined query	Num
$R_1^{conquer^s}$	$\pi_{name}\sigma_{year\geq 1965 \wedge pts\geq 2302}$ $(Player \bowtie Regular)$	1
$R_1^{conquer}$	$\pi_{name}\sigma_{year\geq 1965 \wedge pts\geq 2302 \wedge dreb\leq 121 \wedge asts\geq 282 \wedge oreb\leq 507 \wedge weight\geq 165}$ $(Player \bowtie Regular)$	1
$R_1^{talos^+}$	$\pi_{name}\sigma_{pts>2345 \wedge asts>403 \wedge ftm\leq 675 \wedge gp\leq 80 \wedge pf\leq 286 \wedge reb>223}$ $(Player \bowtie Regular)$	1
$R_2^{conquer^s}$	$\pi_{name,team}(Player \bowtie Regular \bowtie TeamSeason)$	1
$R_2^{conquer}$	$\pi_{name,team}\sigma_{turnover\geq 162 \wedge d\_3pa\geq 1191 \wedge pace\leq 93 \wedge blk\geq 26 \wedge weight\leq 210 \wedge d\_dreb\geq 2359 \wedge tpm\leq 88 \wedge o\_asts\leq 1790 \wedge asts\geq 275}$ $(Player \bowtie Regular \bowtie TeamSeason)$	1
$R_2^{talos^+}$	$\pi_{name,team}\sigma_{fga>1526 \wedge o\_reb\leq 3312 \wedge pf\leq 178 \wedge weight>165}$ $(Player \bowtie Regular \bowtie TeamSeason)$	1
$R_3^{conquer^s}$	$\pi_{name,AVG(pts)}\mathcal{G}_{name}\sigma_{year\leq 1970 \wedge pts\geq 2596}$ $(Player \bowtie Regular)$	1
$R_3^{conquer}$	$\pi_{name,AVG(pts)}\mathcal{G}_{name}\sigma_{year\leq 1970 \wedge pts\geq 2596}$ $(Player \bowtie Regular)$	1
$R_3^{talos^+}$	$\pi_{name,AVG(pts)}\mathcal{G}_{name}\sigma_{year\leq 1971 \wedge pts>2637}$ $(Player \bowtie Regular)$	1
$R_4^{conquer^s}$	$\pi_{name,SUM(pts)}\mathcal{G}_{name}\sigma_{year\geq 2007 \wedge pts\geq 2250 \wedge blk\geq 81}$ $(Player \bowtie Regular)$	2
$R_4^{conquer}$	$\pi_{name,SUM(pts)}\mathcal{G}_{name}\sigma_{year\geq 2007 \wedge pts\geq 2250 \wedge blk\geq 81}$ $(Player \bowtie Regular)$	2
$R_4^{talos^+}$	$\pi_{Name,SUM(pts)}\mathcal{G}_{name}\sigma_{year\leq 2003 \wedge first\_season>2002 \wedge ftm>202 \wedge tpm\leq 63}$ $(Player \bowtie Regular)$	1
$R_5^{conquer^s}$	$\pi_{team,SUM(won)}\mathcal{G}_{team}\sigma_{lost\leq 28 \wedge d\_pts\geq 8109 \wedge year\geq 1992}$ $(Player \bowtie Regular)$	1
$R_5^{conquer}$	$\pi_{team,SUM(won)}\mathcal{G}_{team}\sigma_{lost\leq 28 \wedge d\_pts\geq 8109 \wedge year\geq 1992 \wedge d\_fgm\leq 3139 \wedge o\_blk\geq 410}$ $(Player \bowtie Regular)$	1
$R_5^{talos^+}$	$\pi_{team,SUM(won)}\mathcal{G}_{team}\sigma_{o\_fga\leq 3989 \wedge d\_fgm\geq 1708 \wedge d\_oreb\leq 628}$ $(Player \bowtie Regular)$	1

Table 5: Refined queries for test queries on Basketball data set

selection predicates as follows: $year \geq 2007$, $pts \geq 2250$, and $blk \geq 81$; and its output now contains (Wade, 2386) and (James, 4554).

Query Q_5 computes the total games won by teams that satisfy the following conditions: $lost < 30$, $dpts > 8000$, and $year \geq 2008$. The result contains two tuples, ("DEN", 108) and ("LAL", 65). The complex why-not question W_5 asks why the team "CHI" is not in the result such that among the three teams, (1) the total games won by "CHI" is the minimum, and (2) the total games won by "LAL" becomes the maximum. The refined query computed by `ConQueR` modifies the predicates as follows: $lost \leq 28$, $dpts \geq 8109$ and $year \geq 1992$; and its output now contains the tuples ("CHI", 57), ("DEN", 108), and ("LAL", 122).

8. CONCLUSION

In this paper, we have proposed a new paradigm for explaining why-not questions on query results. Our approach, named `ConQueR`, is based on automatically generating refined queries as a means to explain why-not questions. We have proposed novel algorithms to generate good quality refined queries that are not only similar to the original query but also produce precise query results with minimal irrelevant tuples. Besides the basic SPJ queries, `ConQueR` can also answer complex why-not questions on SPJ queries with aggregation that involve comparison constraints. Our experimental results demonstrated that `ConQueR` not only offers a more flexible approach to explain why-not questions, but its constraint-based method of deriving refined queries is also more efficient than an existing classification-based method.

9. REFERENCES

[1] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[2] A. Chapman and H. V. Jagadish. Why not? In *SIGMOD*, pages 523–534, 2009.

[3] W. W. Chu and Q. Chen. A structured approach for cooperative query answering. *IEEE Trans. on Knowl. and Data Eng.*, 6(5):738–749, 1994.

[4] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. In *VLDB*, pages 471–480, 2001.

[5] R. Dhamankar, Y. Lee, A. Doan, A. Halevy, and P. Domingos. iMAP: discovering complex semantic matches between database schemas. In *SIGMOD*, pages 383–394, 2004.

[6] M. Herschel, M. A. Hernández, and W.-C. Tan. Artemis: a system for analyzing missing answers. *PVLDB*, 2(2):1550–1553, 2009.

[7] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. *PVLDB*, 1(1):736–747, 2008.

[8] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. In *SIGMOD*, pages 13–24, 2007.

[9] N. Koudas, C. Li, A. K. H. Tung, and R. Vernica. Relaxing join and selection queries. In *VLDB*, pages 199–210, 2006.

[10] C. Mishra and N. Koudas. Interactive query refinement. In *EDBT*, pages 862–873, 2009.

[11] I. Muslea and T. J. Lee. Online query relaxation via bayesian causal structures discovery. In *AAAI*, pages 831–836, 2005.

[12] S. Sarawagi. Explaining differences in multidimensional aggregates. In *VLDB*, pages 42–53, 1999.

[13] W.-C. Tan. Provenance in databases: Past, current, and future. *IEEE Data Eng. Bull.*, 30(4):3–12, 2007.

[14] Q. T. Tran and C.-Y. Chan. How to ConQueR why-not questions. Technical Report http://www.comp.nus.edu.sg/~tqtrung/conquer-tech.pdf, National University of Singapore, March 2010.

[15] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query by output. In *SIGMOD*, pages 535–548, 2009.